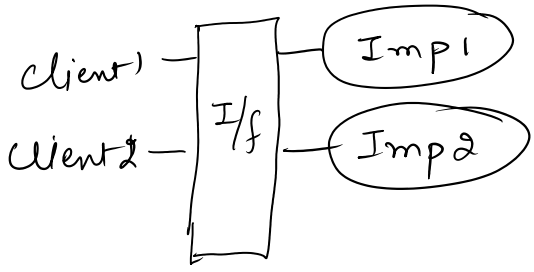
ABSTRACT DATA TYPES.

int – set of values, set of operation +, −, *, /, %, /0

ADT – set of values, Set of Operations



Stack – push() pop()

- push()
- pop()

Array or Linked List

Advantages :–

- code reuse/ modular
- Easy & robust
- change functionality without impacting client's code
- Used to seperate the Data Structure

from details of Implementation

⇒ 5+7 57+ +57

Infix postfix prefix

Evaluate a postfix Expression.

```
int eval_postfix (const char * exp)
{
    Stack stack ( strlen (exp))
    while (* exp)
    {
        char c = * exp
        if ( is_operand (c)) Stack.push(c)
        else if (is_operator (c))
            eval ( c, Stack. pop (), Stack. pop)
        ++exp;
    }
}
```

Main Idea –

1. Operand – Push into stack
2. Operator – OP_1 operator OP_2

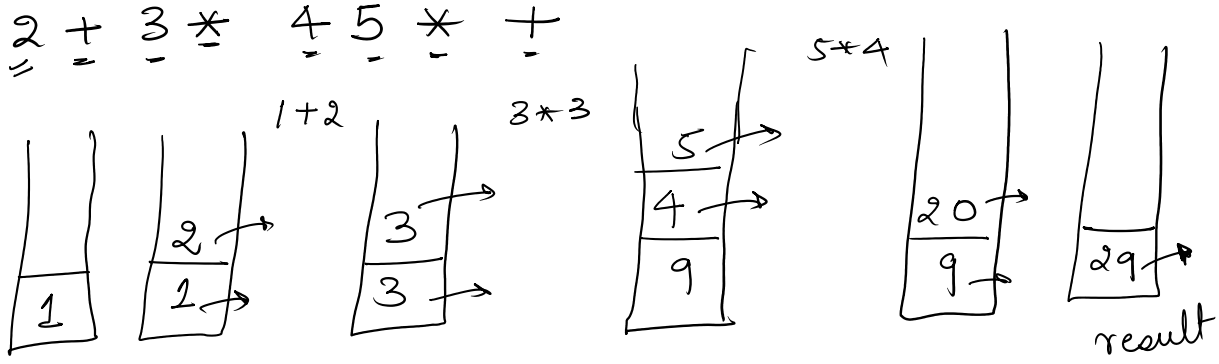
push result into stack

3. pop the top of the stack

⇓

O/P

1 2 + 3 * 4 5 * +



⟹ Convert Infix to Postfix

1. Operand send to Output
2. Left parenthesis - push paranthesis onto the stack
'('
3. Right parenthesis - pop out all the operators send to O/P till you get a left paranthesis
')'
4. Operator
   i) If stack is empty push the operator onto the stack
   ii) If top of stack element is '(' push the operator onto the stack

iii) If top of stack element is an operator with lower or equal precedence push the scanned operator onto the stack

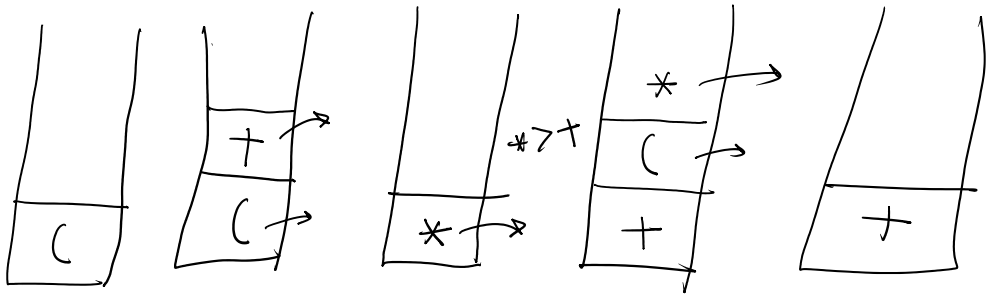
Precedence	
−	1
+	2
*	3
/	4

low ↓ high

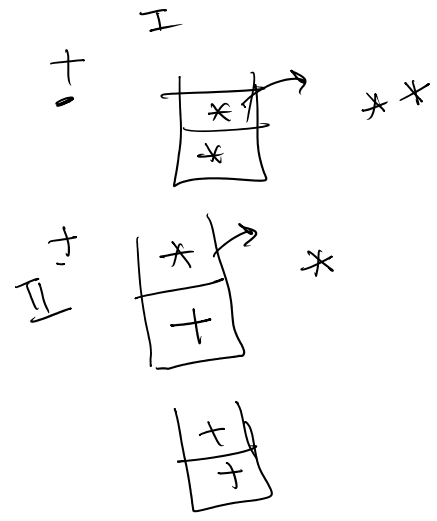
iv) If top of stack element is an operator with higher precedence. Pop out top of the Stack operator. Send it to Output. check with the new top of the stack ~~Push the scanned operator onto the stack~~

Pop out all the operators once exp is finished

(1+2) * 3 + (4*5)



12+3*45*+



Stack - char items

Stack (int capacity)

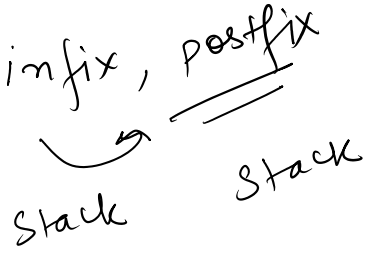
void Push (char item)

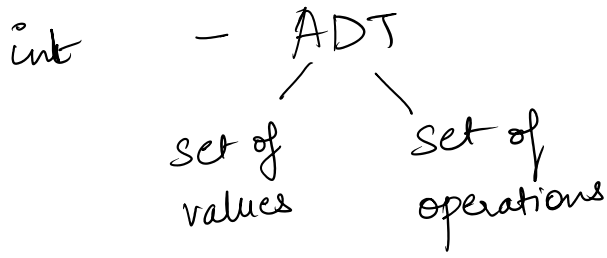
```
char Pop ()
bool  Is Empty ()

class Stack {
    char * items;  // Item * items
    int size.
    int capacity

public:
    Stack (int capacity): capacity {capacity}
    {
        items = new char [capacity]
    }                    Item
    ~ Stack () {
        delete [] items          // O(n)
    }
    void Push (char item)        //O(1)
    {   if (size < capacity)
            items [size++] = item
        else return
    }
    void Pop () {                //O(1)
        return items [size--];
    }
    bool  Is Empty ()            //O(1)
```

```
void resize (int new-capacity)
{
    = new char [new-capacity]
copy all elements from the prev stack to the resized
```

```
bool IsEmpty()  //O(1)
{ if(size == =0) return true }
    else return false
}
}
```

the prev ... to the resized new stack





⇒2 Datatype of var. - not restricted to char

```
template <typename Item>
  Item * items;
```

⇒3 Not fixed size Stack

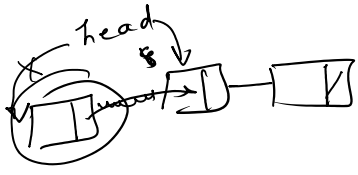
LL - Stack

```
class StackLL: public Stack<Item>
{
   Struct Node
   { Node * next;
     Item data;
   }
public:
```

```
public:
    Stack LL (const int& capacity): (capacity)
    { head = 0;
      size.
    }
```


```
~Stack LL ()
  { while(head)                                  // O(n)
      { Node * n = head * next;
        OA -> free(head)
  } } head = n
```

```
Void Push (const Item & item) override
{ If(size >= capacity) return;                   // Adding in the front of LL
  Node * node = OA. allocate();
  node -> data = item                            // O(1)
  node -> next = head
  head = node
  ++ size
}
```



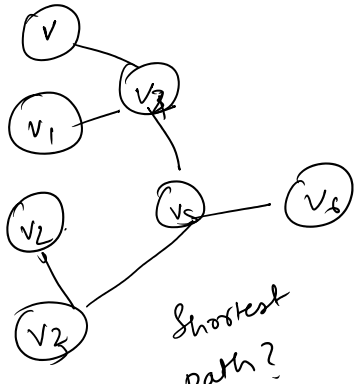

```
Item Pop() override {
  Item item = head -> data                       // O(1)
  Node * temp = head
```

```
head = head → next
OA. free(temp)
return item
}

bool isEmpty() override {
return (head == 0);        // O(1)
}

void resize (int new_capacity)
{
    capacity = new_capacity        // O(1)
}
```

QUEUES. — FIFO

Array Representation

```
Queue <vertex> queue {vertex}[n]
while(!queue.isEmpty()){
    vertex v = queue.pop()
    analyse(v)
    for each (vn: v.getAdjacent())
        queue.push(vn)
}
template < typename Item>
```

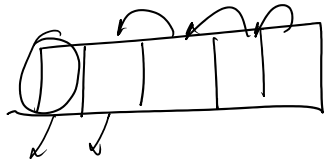


```
class Queue Arr : Queue<Item> {
  public Queue (unsigned capacity):
    capacity_ {capacity}, size_{0} {
      items_ = new Item [capacity];
  }
  ~ Queue Arr() {
      delete. [] items
  }
  void Push (const Item & item) override {
    if( size_ >= capacity) // checking full
      return
    items_ [size] = item;
    ++ size
}
Item Pop () override {
    if ( IsEmpty()) // return
    item = items_ [0]                    // O(n)
    for (i = 0; i < size - 1; ++i)
    { items_ [i] = item_[i+1]
    }
      -- size;
```

```
    --size;
}
return item
```

⇒ Linked List

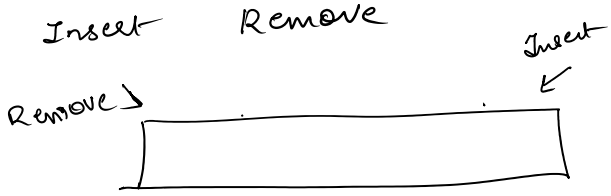
Dynamic size

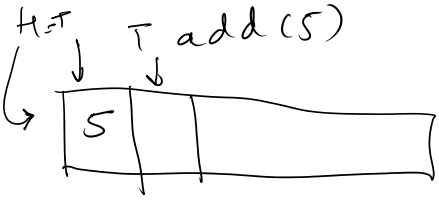


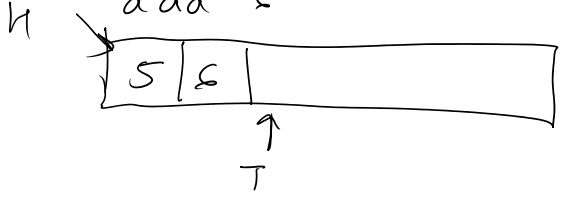
Circular Array

Head Tail

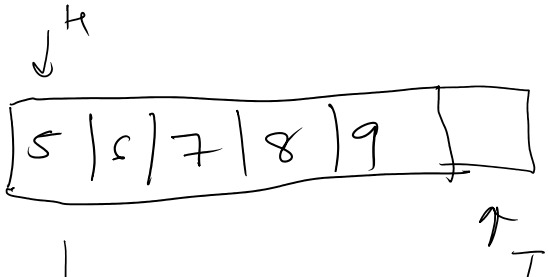
H T







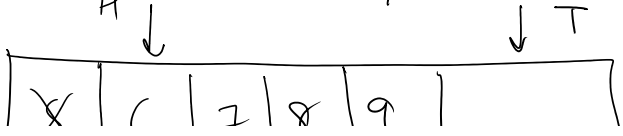
add(7) add(8)
add(9)

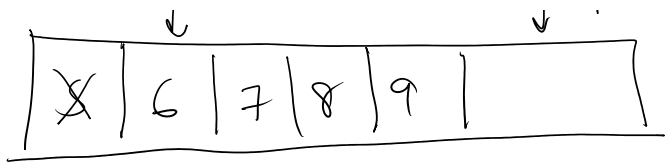


Removing - Location of H

Adding - Location of T

remove() / pop()





H == T // Q is empty

(T+1) % size == H // Q is full

items in Q = (T - H + size) % size

⇒ Priority Queues - extract / removal of an element is based on a priority